

Department I - C Plus Plus

Modern and Lucid C++ for Professional Programmers

Week 1 – Introduction to C++

Thomas Corbat / Felix Morgner
Rapperswil, 20.09.2022
HS2022



- You become motivated to learn C++
- You get familiar with the basics of C++
- You understand how the C++ toolchain processes header and source files
- You can setup a project with separated unit tests and executable for a library
- You understand how declarations and definitions work in C++
- You can explain the value semantics of C++

- **Module Overview**
- **Why C++**
- **Introduction**
- **C++ Compilation Process**
- **Modularization & Testing**
- **Declarations and Definitions**
- **Syntax: Java vs. C++**

Module Overview



- **Main platform is gitlab**

- <https://gitlab.ost.ch/cxx/cpl>
- It contains a repository you can clone using git

- **Each week features**

- Lecture slides
- Exercises (online and as PDF)
- Solutions (except Testat and first week)

- **Obsolete lecture video material is available on SwitchTube (for C++17)**

- <https://tube.switch.ch/channels/889a82a4>

- **You will need to hand-in 2 of 3 testat exercises**
- **Testat 1 and 2 consist of exercise task from multiple weeks**
- **You get feedback for your solution**
 - Pass or fail
 - Detail feedback for your code
 - You need to pass 2 testat hand-ins get allowance for the exam
- **We encourage groups of 2 to 3 students (maximum is 3)**

Week	Topic	Lecturer
1	Introduction	Corbat
2	Values and Streams	Corbat
3	Sequences and Iterators	Corbat
4	Functions and Exceptions	Corbat
5	Classes and Operator Overloading	Corbat
6	Namespaces and Enums	Corbat
	No Lecture due to Holiday	
7	Standard Containers	Corbat
8	Standard Algorithms	Morgner
9	Function Templates	Morgner
10	Class Templates	Morgner
11	Heap Memory	Morgner
12	Dynamic Polymorphism, Variant	Morgner
13	Initialization and Aggregates	Morgner

- You can use whatever IDE you want
- We recommend Visual Studio Code
 - Expect more struggle with the IDE (whichever you use) than with what you might expect from Java
- The C++ Unit Testing Easier (CUTE) Framework is used throughout this lecture
 - www.cute-test.com
 - You need to get familiar with its features
- Boost libraries
 - www.boost.org
- We used to have Cevalop as primary IDE, but it is not ready for C++20
- Other IDEs are possible as well CLion, Visual Studio, ...



- **In the first lecture you will setup your environment**
- **We recommend a recent version of GNU g++ or clang**
 - Linux: g++ (version ≥ 11) / clang (version ≥ 14)
 - Windows: MinGW g++ (version ≥ 11) - <https://nuwen.net/mingw.html> MSYS2 (with MinGW)
 - MacOS: g++ / clang (current version)
 - You might need to adapt your PATH variable to contain the compiler executable
- **If you doubt your compiler is correct, you can check an online-compiler for a different opinion**
 - <https://godbolt.org/>

Why C++

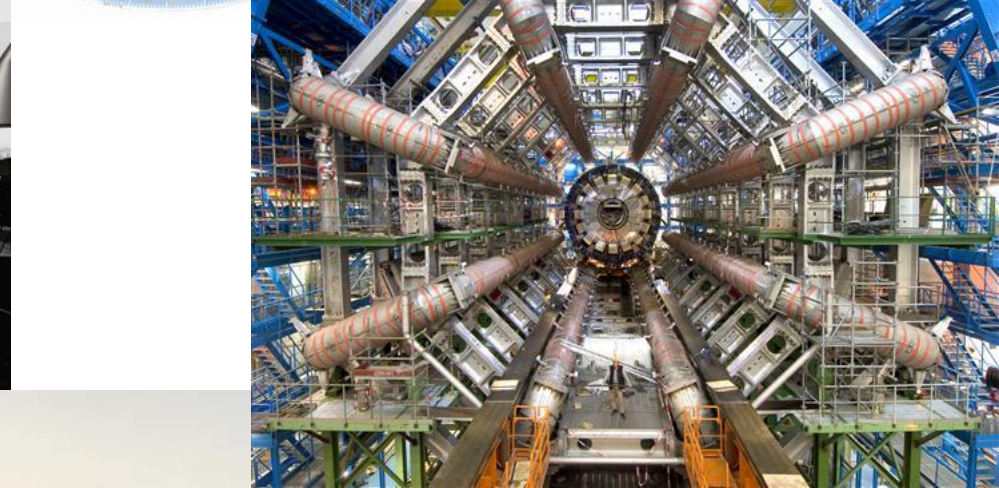
Goals:

- You are motivated to learn C++



Why C++?

11



- **ISO standard: C++20 (C++23 forthcoming)**

- Parts of it designed and decided at HSR (August 2010; June 2014; June 2018)
- Works on almost all platforms: micro controller to main frame

- **Multi-paradigm language with zero-cost abstraction**

- Not just object-oriented -> more mechanisms for abstraction!

- **High-level abstraction facilities**

- Write less code, but need to know what actually happens!

- **Cool stuff to learn, valuable also for other languages!**



- **C++20 as of ISO 14882 (but not all: 1866 p.)**
- **Modern C++ usage, not "classic" 1990's style**
 - Much simpler, especially from 2011 ISO standard on
- **Effective use of Standard Library (STL)**
 - Avoiding re-inventing loops or containers
- **Modern Unit Testing with CUTE**
 - Not having test automation is unprofessional
- **Applied software engineering practices**

Introduction to C++



Goals:

- You get familiar with the basics of C++

- `main()` is the program entry function
 - The "Big Bang" in the execution of every program
- **C++ provides functions (not methods as Java)**
 - Not all functions are bound to a class or object! the latter are called "**member-functions**" not methods
- **Return types are written in front of the function name (most of the time) in declarations**
 - or `auto` if the return type should or can be deduced (main's return type must be `int`)
- **Parentheses after the name `()` mark a function**
- **Braces `{ }` mark the function body**
- **Implicitly returns 0**

```
int main() {  
}
```

```
auto main() -> int {  
}
```

- Hello World program as generated by Eclipse CDT

```
//=====
// Name      : HellWorld.cpp
// Author    :
// Version   :
// Copyright  : Your copyright notice
// Description: Hello World in C++, Ansi-style
//=====

#include <iostream>
using namespace std;

int main() {
    cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!!
    return 0;
}
```

Belongs into a version management system

Bad practice, very bad in global scope

Ridiculous comment

Redundant

Inefficient and redundant

Using a global variable! Really bad (except in main())

- **Compiler error messages can be hard to understand**
- **One can program C and call it C++**
- **There is no garbage collection like in Java or .NET**
 - But you don't need it if you program like we teach you!
- **IDE support is not yet up to Java**
- **Compilers might behave different than specified by the standard!**
- **C++ standard defines "undefined behavior"**



Undefined behavior

-- behavior, upon use of a non-portable or erroneous program construct, ... for which the standard imposes no requirements. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to having demons fly out of your nose."

- John F. Woods

<http://groups.google.com/groups?hl=en&selm=10195%40ksr.com>

- Our symbols to warn about lurking undefined behavior:



C++ Compilation Process



Goal:

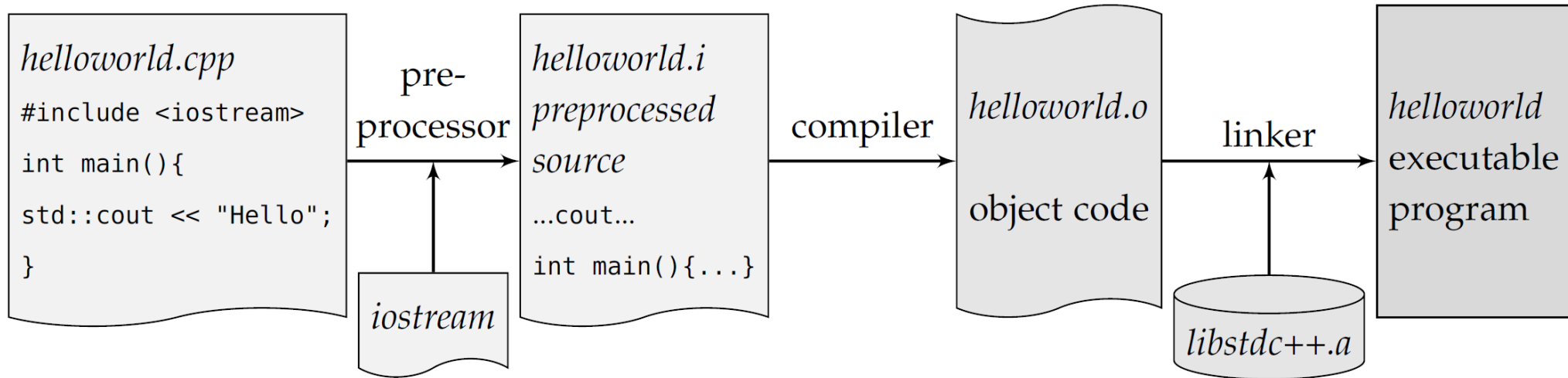
- Repetition from Bsys1
- You understand how the C++ toolchain processes header and source files

- ***.cpp files for source code**

- Also called "Implementation File"
- Function implementations (can be in .hpp files as well)
- Source of compilation - aka "Translation Unit"

- ***.hpp (or *.h) files for interfaces (and templates)**

- Also called "Header File"
- Declarations and definitions to be used in other implementation files
- Textual inclusion through a pre-processor (C++20 has a "Module" mechanism, but tooling is incomplete)
- `#include "header.hpp"`
 - like in C (you know that from Bsys1)



- **C++ is usually compiled into machine code**

- No (Java) Virtual Machine overhead – a bit less flexibility built-in

- **3 Phases of compilation**

- Preprocessor – Textual replacement of preprocessor directives (*#include*)
- Compiler – Translation of C++ code into machine code (source file to object file)
- Linker – Combination of object files and libraries into libraries and executables

main.cpp

```
#include "sayhello.hpp"
#include <iostream>

auto main() -> int {
    sayHello(std::cout);
}
```

sayhello.hpp

```
#ifndef SAYHELLO_HPP_
#define SAYHELLO_HPP_

#include <iosfwd>

auto sayHello(std::ostream&) -> void;

#endif /* SAYHELLO_HPP_ */
```

sayhello.cpp

```
#include "sayhello.hpp"
#include <ostream>

auto sayHello(std::ostream& os) -> void {
    os << "Hi there!\n";
}
```

main.cpp

```
#include "sayhello.hpp"
#include <iostream>

auto main() -> int {
    sayHello(std::cout);
}
```

sayhello.hpp

```
#ifndef SAYHELLO_HPP_
#define SAYHELLO_HPP_

#include <iosfwd>

auto sayHello(std::ostream&) -> void;

#endif /* SAYHELLO_HPP_ */
```

Preprocessor

main.i

```
<content of iosfwd>

auto sayHello(std::ostream&) -> void;

<content of iostream>

auto main() -> int {
    sayHello(std::cout);
}
```

main.i

```
<content of iosfwd>

auto sayHello(std::ostream&) -> void;

<content of iostream>

auto main() -> int {
    sayHello(std::cout);
}
```

Compiler

main.o

```
010110101... (machine code)

<Definition of main() which calls
sayHello>
```


main.o

010110101... (machine code)
<Definition of main() which calls
sayHello>

sayhello.o

010110101... (machine code)
<Definition sayHello() which
writes to the ostream parameter>

Linker



sayhello Executable

010110101... (machine code)
<executable program>

Modularization & Testing



Goals:

- You know how and why to separate functionality into a library
- You setup a project with separated unit tests and executable for a library

- **Library functions in separate compilation units**

- Allows unit testing

- Declarations in header files

- **Using the library function requires #include**

- **NOTE: Use "static library project" in**

- shared libraries can introduce unnecessary hassle

```
#include "sayhello.hpp"
#include <ostream>
auto sayhello(std::ostream & out) -> void {
    out << "Hello world!\n";
}
```

```
#ifndef SAYHELLO_HPP_
#define SAYHELLO_HPP_

#include <iosfwd>
auto sayhello(std::ostream & out) -> void;

#endif /* SAYHELLO_HPP_ */
```

```
#include "sayhello.hpp"
#include <iostream>
auto main() -> int {
    sayhello(std::cout);
}
```

C++ Library

```
#include "sayhello.hpp"
#include <ostream>

auto sayhello(std::ostream & out) -> void {
    out << "Hello world!\n";
}
```

```
#ifndef SAYHELLO_HPP_
#define SAYHELLO_HPP_

#include <iosfwd>

auto sayhello(std::ostream & out) -> void;

#endif /* SAYHELLO_HPP_ */
```

CUTE Test Executable

```
#include "sayhello.h"
#include "cute/cute.h"
#include "cute/ide_listener.h"
#include "cute/cute_runner.h"
#include <sstream>

auto testSayHelloSaysHelloWorld() -> void {
    std::ostringstream out{};
    sayhello(out);
    ASSERT_EQUAL("Hello world!\n", out.str());
}

auto runAllTests() -> void {
    cute::suite suite{ };
    suite.push_back(CUTE(testSayHelloSaysHelloWorld));
    cute::ide_listener<> lis{};
    auto const runner = cute::makeRunner(lis);
    runner(suite, "All Tests");
}

auto main() -> int {
    runAllTests();
}
```

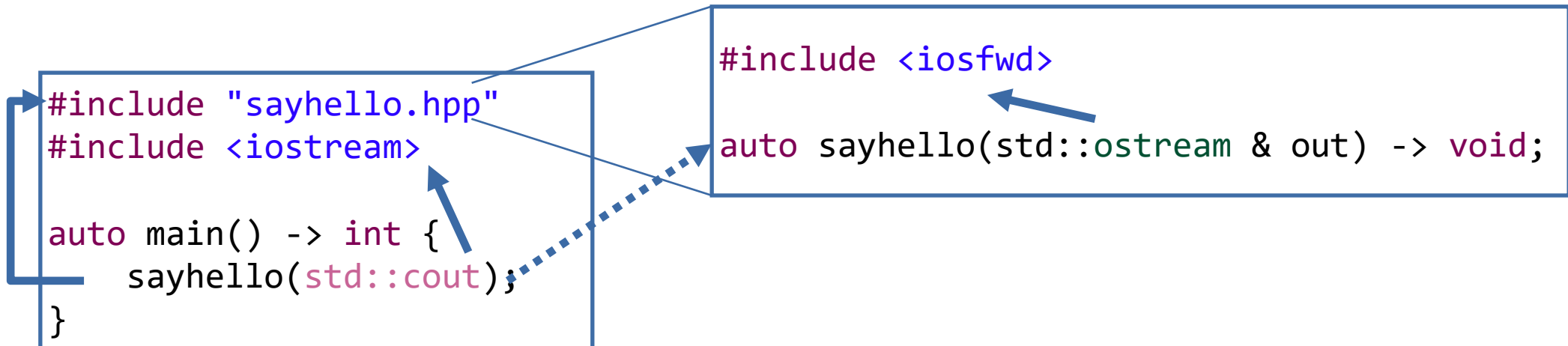
Declarations and Definitions



Goals:

- You know the difference between declarations and definitions
- You know about the One Definition Rule

- All things with a name that you use in a C++ program must be declared before you can do so
 - E.g. a function that you call (major difference from C)
 - A type that you use for a variable (except some built-ins)
 - A variable that you use
- For library code such declarations are put in header files
 - `#include <iostream>` to use `std::cout`



```
auto          <function-name>(<parameters>) -> <return-type>;
<return-type> <function-name>(<parameters>);
```

- **Tells the compiler that there is a function named <function-name> that takes the parameters <parameters> and returns a value of type <return-type>**

Term	Description
Return Type	Every function either returns a value of a specified type or it has return type void
Function Name	Identifier for the function. There can be multiple functions with the same name. Different parameter types are required (Function Overloading)
Parameters	A list of 0 to N parameters. Each parameter has a type and an optional name.
Signature	Combination of name and parameter types. Used for overload resolution (distinction between functions with the same name)

- **Declarations are usually put into a header file (*.hpp)**
- **There can be multiple declarations of the same function**

```
auto          <function-name>(<parameters>) -> <return-type> { /*body*/}  
<return-type> <function-name>(<parameters>)                  { /*body*/}
```

- **Implementation of a function**

- Specifies what the function does

Term	Description
Return Type Function Name Parameters Signature	Same as for function declaration
Body	Implementation of the function with 0 to N statements

- **Definitions are usually put into a source file (*.cpp)**
- **There can only be one definition of the same function (One Definition Rule)**


```
#include "sayhello.hpp"
#include <ostream>

auto sayhello(std::ostream & out) -> void {
    out << "Hello world!\n";
}
```

- If a function has a non-void return type it must return a value on every path (or throw an exception)

```
#include <stdexcept>


auto divide(int dividend, int divisor) -> int {
    if (divisor == 0) {
        throw std::invalid_argument{"Divisor must not be 0."};
    }
    return dividend / divisor;
}
```

- While a program element can be declared several times without problem there can be only one definition of it
- This is called the One Definition Rule (ODR)!
- Consequences:
 - There can be only one definition of the `main()` function
 - Or any other function with the same signature
 - There must be a definition for all elements that are used
- **#include guards recommended**
 - not required for function declarations
 - but for (class) type definitions in header files

```
#include "sayhello.h"
#include <iostream>

auto main() -> int {
    sayhello(std::cout);
}

auto main() -> int{
    saygoodbye(std::cout);
}
```



- Include guards ensure that a header file is only included once
- Multiple inclusions could violate the One Definition Rule when the header contains definitions

sayhello.hpp

```
#ifndef SAYHELLO_HPP_  
#define SAYHELLO_HPP_  
#include <iosfwd>  
struct Greeter {  
    //...  
};  
#endif /* SAYHELLO_HPP_ */
```

Be aware to adjust
when copying or
renaming a header file

Directive	Description
#ifndef SYMBOL	Checks whether SYMBOL has already been defined If not the block until #endif is included
#define SYMBOL	Defines SYMBOL
#endif	Closes the block opened by #ifndef

C++ Is NOT Java

Goals:

- You know about the existence of false friends between C++ and Java



- **Beware! Java knowledge may be the wolf in sheep's clothing!**

- Syntax of Java is based on C++'s syntax, which in turn is based on C's syntax

```
public class HelloWorld {  
    public static void main (String[] args) {  
        System.out.println("Hello World");  
    }  
}
```



```
#include <iostream>  
auto main() -> int {  
    std::cout << "Hello World\n";  
}
```



```
#include <stdio.h>  
int main() {  
    puts("Hello World\n");  
}
```

Bad Code

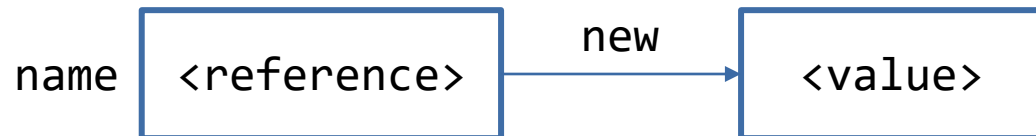
- **More C++ vs Java in self-study slides**

- **Java's objects are placed on the heap**
 - Exception: Primitive values (int, float, boolean)
- **C++ allocates memory for variables on definition**
 - No **explicit** heap memory needed
 - No indirection and space overhead


```
Type name = new Type();
```




```
Type name{};
```



```
public class Point {  
    private int x;  
    private int y;  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public static void main(String[] args) {  
        Point point = new Point(1, 20);  
        Point samePoint = point;  
        point.x = 300;  
        System.out.println(samePoint.x);  
    }  
}
```

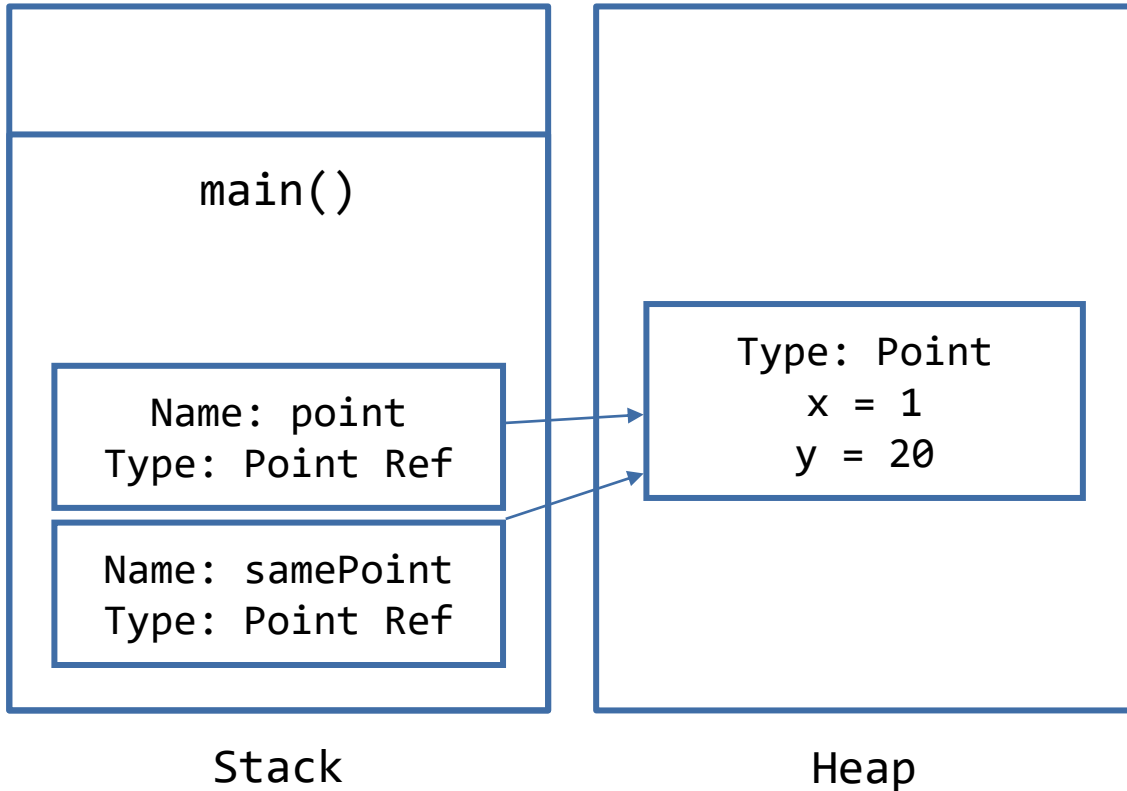


```
#include <iostream>  
  
struct Point {  
    int x;  
    int y;  
};  
  
auto main() -> int {  
    Point point{1, 20};  
    Point otherPoint{point};  
    point.x = 300;  
    std::cout << otherPoint.x << '\n';  
}
```

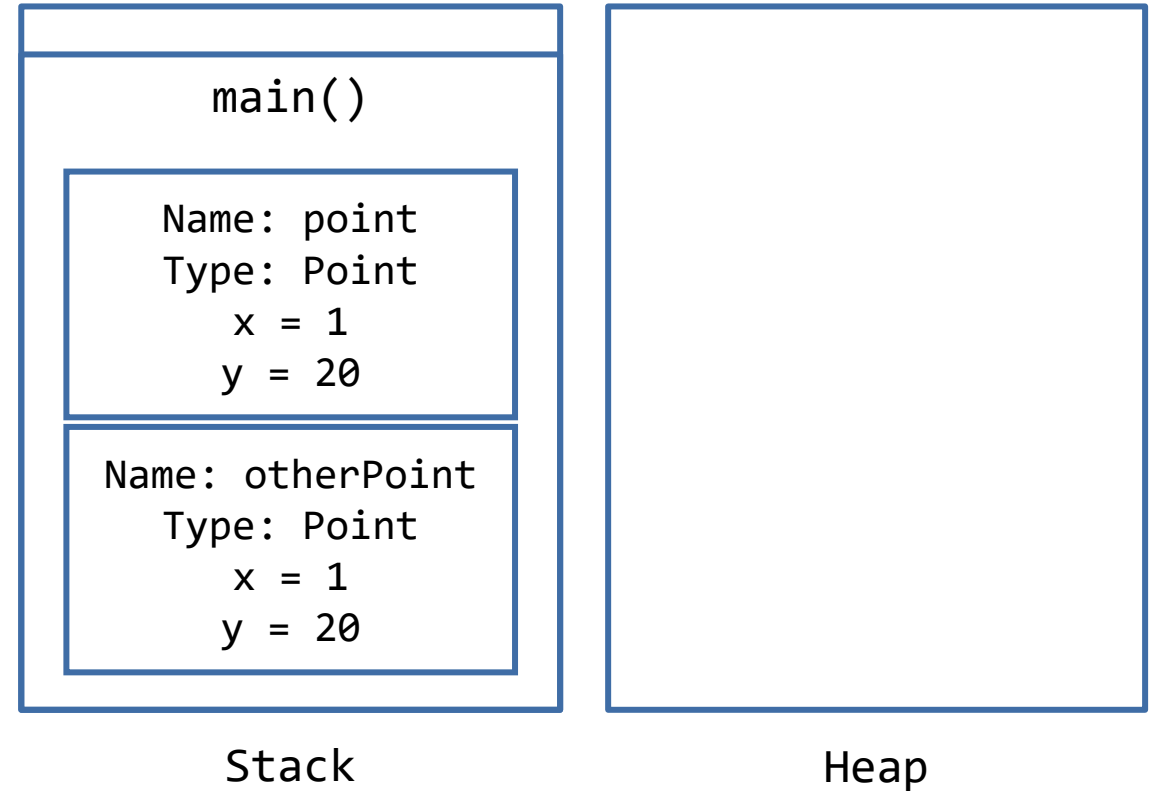




```
public static void main(String[] args) {  
    Point point = new Point(1, 20);  
    Point samePoint = point;  
    point.x = 300;  
    System.out.println(samePoint.x);  
}
```



```
auto main() -> int {  
    Point point{1, 20};  
    Point otherPoint{point};  
    point.x = 300;  
    std::cout << otherPoint.x << '\n';  
}
```





```
public static void main(String[] args) {  
    Point point = new Point(1, 20);  
    Point samePoint = point;  
    point.x = 300;  
    System.out.println(samePoint.x);  
}
```

main()

Name: point
Type: Point Ref

Name: samePoint
Type: Point Ref

Type: Point
x = 300
y = 20

Stack

Heap



```
auto main() -> int {  
    Point point{1, 20};  
    Point otherPoint{point};  
    point.x = 300;  
    std::cout << otherPoint.x << '\n';  
}
```

main()

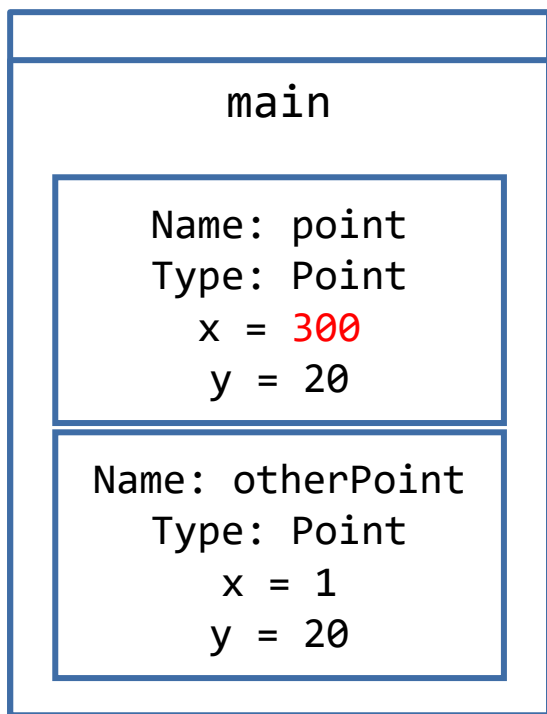
Name: point
Type: Point
x = 300
y = 20

Name: otherPoint
Type: Point
x = 1
y = 20

Stack

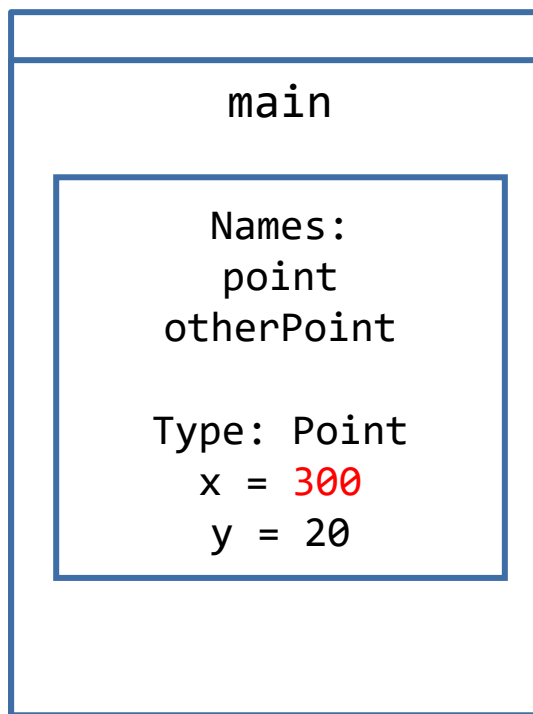
Heap

```
auto main() -> int {  
    Point point{1, 20};  
    Point otherPoint{point};  
    point.x = 300;  
    std::cout << otherPoint.x << '\n';  
}
```



Stack

```
auto main() -> int {  
    Point point{1, 20};  
    Point & otherPoint{point};  
    point.x = 300;  
    std::cout << otherPoint.x << '\n';  
}
```



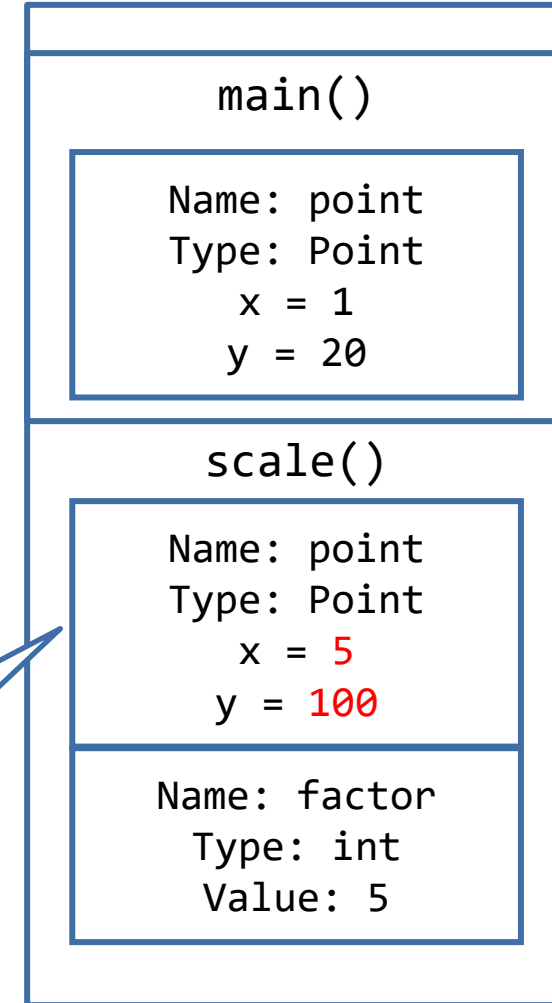
Stack

- No side-effect on the call-site

```
auto main() -> int {  
    Point point{1, 20};  
    scale(point, 5);  
}  
  
auto scale(Point point, int factor) -> Point {  
    point.x *= factor;  
    point.y *= factor;  
    return point;  
}
```

Copy

scale() has its own
copy of argument



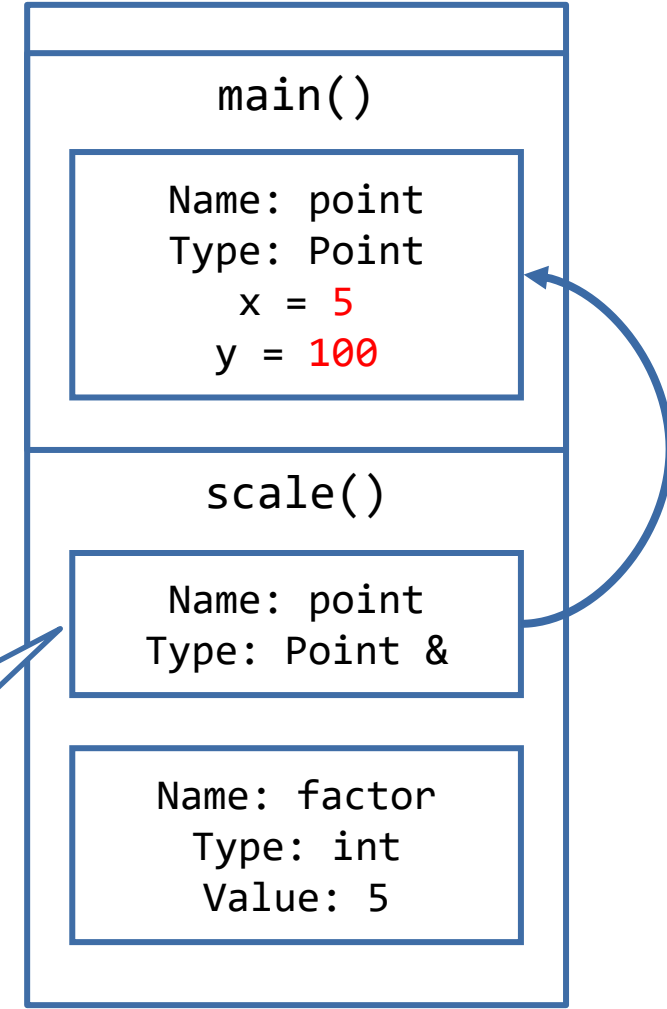
Stack

- Side-effect on the call-site

```
auto main() -> int {  
    Point point{1, 20};  
    scale(point, 5);  
}  
  
auto scale(Point & point, int factor) -> void {  
    point.x *= factor;  
    point.y *= factor;  
}
```

Reference

Uses original point
argument in the
scale() function



Stack

- **C++ is substantially different from Java**
 - even though this is not obvious regarding the similarities in the syntax (Java borrowed C++ syntax)
 - Most important difference is the existence of Undefined Behavior
- **The translation process consists of three main stages (preprocessor, compiler and linker)**
- **C++ function code is separated into declarations (in header files) and definitions (in source files)**
- **The One Definition Rule ensures there is only a single definition for each variable, function and type**
 - violating the ODR can result in Undefined Behavior!
- **Modularization of code enables proper unit testing**